



## 저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Master's Thesis

# A Compression Technique Exploiting References for Data Synchronization Services

Wooseung Nam

Department of Computer Science and Engineering

Graduate School of UNIST

2019

# A Compression Technique Exploiting References for Data Synchronization Services

Wooseung Nam

Department of Computer Science and Engineering

Graduate School of UNIST

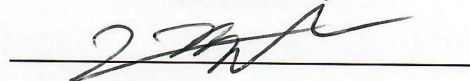
# A Compression Technique Exploiting References for Data Synchronization Services

A dissertation  
submitted to the Graduate School of UNIST  
in partial fulfillment of the  
requirements for the degree of  
Master of Engineering

Wooseung Nam

1.11.2019

Approved by



Advisor

Kyunghan Lee

# A Compression Technique Exploiting References for Data Synchronization Services

Wooseung Nam

This certifies that the dissertation of Wooseung Nam is approved.

1.11.2019

signature



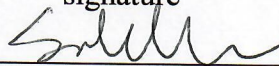
Kyunghan Lee

signature



Changhee Joo

signature



Sungahn Ko

## Abstract

In a variety of network applications, there exists significant amount of shared data between two end hosts. Examples include data synchronization services that replicate data from one node to another. Given that shared data may have high correlation with new data to transmit, we question how such shared data can be best utilized to improve the efficiency of data transmission. To answer this, we develop an encoding technique, SyncCoding, that effectively replaces bit sequences of the data to be transmitted with the pointers to their matching bit sequences in the shared data so called references. By doing so, SyncCoding can reduce data traffic, speed up data transmission, and save energy consumption for transmission. Our evaluations of SyncCoding implemented in Linux show that it outperforms existing popular encoding techniques, Brotli, LZMA, Deflate, and Deduplication. The gains of SyncCoding over those techniques in the perspective of data size after compression in a cloud storage scenario are about 12.4%, 20.1%, 29.9%, and 61.2%, and are about 78.3%, 79.6%, 86.1%, and 92.9% in a web browsing scenario, respectively.



## Table of Contents

1	Introduction.....	ii
2	Related Work.....	3
3	LZMA Primer.....	5
4	System Design and Analysis.....	8
4.1	System Design.....	8
4.2	Comparative Analysis.....	9
4.3	Questions on SyncCoding.....	11
5	Characterization of SyncCoding.....	12
5.1	Reference Selection.....	12
5.2	Maximum Compression Efficiency.....	14
5.3	Referencing Overhead Optimization.....	16
5.4	Encoding Time and Decoding Time of SyncCoding.....	17
5.5	Mobile Energy Consumption of SyncCoding.....	18
6	Evaluation.....	21
6.1	Settings.....	21
6.2	Use Case 1: Cloud Data Sharing.....	22
6.3	Use Case 2: Web Browsing.....	25
7	Discussion.....	28
8	Concluding Remarks.....	30



## List of Figures

1. Sample encoding of (a) LZ77 and (b) LZMA over a sequence of symbols. Whenever a match exists, the longest match is encoded with a length-distance pair. No match lets the symbol be encoded. When there is a distance value repeated recently, LZMA points to it instead of directly encoding it. -----5
2. The concept and basic operations of SyncCoding. -----8
3. The compression ratios of LZMA and SyncCoding with one reference whose modified cosine similarity is ranked by either of Boolean, Log, and Linear. Overall, SyncCoding with a single reference shows higher compression ratios than LZMA and a reference of a higher rank achieves a better compression ratio. -----12
4. The compression ratios of SyncCoding with increasing number of references that are selected randomly, by a greedy search, or by the modified cosine similarity rank with either of Boolean, Log, or Linear. In this figure, the overhead for reference indexing is not considered to focus on understanding the impact of reference selection. -----13
5. The compression efficiency of SyncCoding for an increasing number of references. The per-reference overhead is chosen as either of 10 or 20 bytes. -----14
6. Maximum compression efficiencies of SyncCoding obtained from 100 randomly chosen documents are compared with the compression efficiencies of SyncCoding that use only 24 references. Only a little gap exists. -----15
7. The referencing overhead by indexing references with Huffman coding when referencing frequencies of reference candidates are updated over transmissions. -----16
8. Experimental evaluation of  $T_E(x, k)$  (top) and  $T_D(x, k)$  (bottom), the time durations to encode and to decode a file  $x$  with  $k$  references under Intel i7-3770 CPU (3.40 GHz). --17
9. Our test environment for measuring energy consumption of receiving data with or without SyncCoding in an Android device, Galaxy Note 5. Measurements are conducted by a digital power monitor from Monsoon [31]. -----18
10. Fig. 10. The energy consumption measurement results on Galaxy Note 5 smartphone for downloading and decoding data of variable sizes with SyncCoding and LZMA. -----19
11. Overview of the evaluation scenarios: 1) cloud data sharing (left) and 2) web browsing (right). -----21
12. Compression efficiencies of SyncCoding and other techniques (a) for various document sizes to encode, (b) for various numbers of references. (c) A comparison of compressed sizes

of 50 target documents when $k^*$ references are used. -----	22
13. Fig. 13. Compression ratio and compression efficiency of Deduplication without overhead and with overhead for various chunk sizes in the cloud data sharing scenario with 100 reference files. -----	23
14. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes. -----	24
15. Compression efficiencies of SyncCoding, SyncCoding-Cached and three other encoding techniques for the webpages sequentially visited by sample visit histories obtained from (a) CNN (Politics section) and (b) Yahoo (Science section). (c) A comparison of the average compressed sizes of webpages from three websites with no section restriction. -----	25
16. Compression ratio and compression efficiency of Deduplication without and with overhead for various chunk sizes on a CNN webpage with 10 reference pages. -----	25
17. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes. -----	25
18. The compression gain of SyncCoding over LZMA with 90% confidence intervals for three encryption algorithms and for no data encryption. -----	28

## 1 Introduction

During the last decade, cloud-based data synchronization services for end-users such as Dropbox, OneDrive, and Google Drive have attracted a huge number of subscribers. These new services now become indispensable and occupy a huge portion of Internet bandwidth. Given the rise of data synchronization services in which significant amount of shared data exists in the storage of server and client (i.e., end hosts), we raise the following question: *“how the previously synchronized data between the end hosts can be exploited for the delivery of new data between them?”*

We find that this question is not only important to synchronization services but also to general network applications including web browsing and data streaming because data transfer between end hosts essentially lets them have the same data (i.e., synchronized data). Unfortunately, this question has been under-explored in the literature and has not been well addressed in practical systems. Index code [2], which suggested the concept of encoding a block of data with its relations to other blocks of data, the first of its kind, but did not consider how to select subsets of blocks to improve efficiency. Deduplication [3], which finds duplicated data chunks in a storage system for the elimination of redundancy is related but because Deduplication mostly works at the level of files or bit chunks of a fixed size (e.g., 4MB in Dropbox) in a local storage, its ability to utilize the shared data over the network is limited.

In this paper, we systematically answer this question by proposing a new data encoding technique called SyncCoding that can be used for data synchronization services with shared data. The intuition behind SyncCoding is to quickly choose a set of data from the previously synchronized data, which holds high similarity with the new data to be synchronized, and to encode the new data by intelligently fragmentizing it into bit sequences that can be referenced from the chosen set. By its nature, SyncCoding works effectively with the types of data that are created on similar topics, directed in similar formats, or authored by the persons of similar writing styles. We find it interesting that a large portion of data being handled by data synchronization services such as documents on the same topic collected in a folder of a cloud storage, web pages of a website authored by a programmer, and technical reports in a given format fall into the category where SyncCoding can be effective.

SyncCoding demonstrates a way to efficiently utilize the synchronized data and gives a quantitative answer on how helpful the synchronized data can be for compressing data. In order to do so, we take the following steps.

- 1) We revisit the algorithm of LZMA (Lempel-ZivMarkov chain algorithm) [4], the core of 7-zip compression format [5] that is known as one of the most popular data encoding techniques and reveal how it works in detail.

- 2) We design the work flow of SyncCoding using the ideas from LZMA under the existence of previously synchronized data, called potential *references*<sup>1</sup>.
- 3) We analyze in what conditions SyncCoding outperforms LZMA in the size of compressed data and suggest practical heuristic algorithms to select actual references from the potential references in order to meet the conditions.
- 4) We implement SyncCoding in a Linux system and evaluate its compression characteristics. We also implement it in an Android system and study its energy consumption characteristics. We further demonstrate the benefits of using SyncCoding in realistic use cases of cloud data sharing and web browsing.
- 5) We study the performance of SyncCoding for encrypted data, and discuss an implementation guideline for SyncCoding.

Our extensive evaluation of SyncCoding in the cloud data sharing scenario with a dataset of RFC (Request For Comments) technical documents reveals that SyncCoding compresses on average a document about 10.6%, 20.1%, and 61.2% more compared to LZMA after Deduplication, LZMA without Deduplication, and Deduplication only, about 17.2% and 30.9% more compared to Deflate with and without Deduplication, and about 7.6% and 12.4% more compared to Brotli with and without Deduplication, respectively. Upon further evaluation of SyncCoding in the web browsing scenario shows that SyncCoding outperforms commercial web speed-up algorithms, Brotli from Google [7] with and without Deduplication by 79.4% and 83.2%, and Deflate [8] with and without Deduplication, by 83.9% and 87.0%, respectively, in the size of compressed webpages of CNN. We find that this substantial gain captured by SyncCoding comes from the similar programming style maintained over the webpages of the same website and confirm that the gain is persistent over various websites such as CNN and NY Times.

---

<sup>1</sup> The work flow of SyncCoding is not dependent on a specific compression algorithm as its main structure is about efficiently exploiting inter-data correlation for compression. Therefore, SyncCoding can be easily extended to use more recent Lempel-Ziv based algorithms or more advanced algorithms such as PAQ [6] and its variants.

## 2 Related Work

Reforming a given bit sequence with a new bit sequence to reduce the total number of bits is called data compression and it is also known as source coding. When the original bit sequence can be perfectly recovered from the encoded bit sequence, it is called *lossless* compression which is the focus of this work. A bit sequence is equivalent to, hence interchangeable with, a symbol sequence where a symbol is defined by a block of bits which repeatedly appears in the original bit sequence (e.g., ASCII code). Shannon's source coding theorem [9] tells us that a symbol-by-symbol encoding becomes optimal when symbol  $i$  that appears with probability  $p_i$  in the symbol sequence is encoded by  $-\log_2 p_i$  bits. It is well known that Huffman coding [10] is an optimal encoding for each symbol but is not for a symbol sequence. Arithmetic coding [11] produces a near-optimal output for a given symbol sequence.

However, when the unit for encoding goes beyond a symbol, the situation becomes much more complicated. An encoding with blocks of symbols that together frequently appear may reduce the total number of bits, but it is unclear how to find the optimal block sizes that give the smallest encoded bits. Therefore, finding the real optimal encoding for an arbitrary bit sequence becomes NP-hard [12] due to the exponential complexity involved in testing the combinations of the block sizes.

LZ77 [13], the first sliding window compression algorithm, tackles this challenge by managing dynamically sized blocks of symbols within a given window (i.e., the maximum number of bits that can be considered as a block) by a tree structure. In a nutshell, LZ77 progressively puts the symbols to the tree as it reads symbols and when there is a repeated block of symbols found in the tree, it replaces (i.e., self-cites) the block with the distance to the block and the block length. This process lets LZ77 compress redundant blocks of symbols.

*Deflate* [8] combines LZ77 and Huffman coding. It replaces matching blocks of symbols with length-distance pairs similarly to LZ77 and then further compresses those pairs using Huffman coding. LZ78 and LZMA are variants of LZ77, of which their encoding methods for length-distance pairs are improved. LZMA is the algorithm used in 7z format of the 7-zip archiver. We will later discuss about the operations of LZMA in detail in Section 3.

Unlike the aforementioned compression algorithms, there exist several techniques that include external information in addition to the source data for encoding. Index code [2] generalizes the broadcast problem with the existence of side information at the receivers and analyzes the properties of the optimal encoding of a new data block under the given side information denoted as a graph that captures the relations among the data blocks. Index code is still far from SyncCoding because it does not focus on how to acquire the side-information, which is challenging in practice.

There are simpler ways of exploiting external information such as Star encoding (\*-encoding) [14] that uses an external static dictionary shared between a server and its client. A similar yet more

efficient approach has been made using Length Index Preserving Transform (LIPT) [15] with an English dictionary having about 60,000 words. *Brotli* [7], one of the latest encoding technique, has a pre-defined shared dictionary of about 13,000 English words, phrases, and other sub-strings extracted from a large corpus of text and HTML documents. Brotli is known to achieve about 20% compression gain over Deflate in the encoding of webpages in a web browser [16]. Exploiting a static shared dictionary is useful in general, but its efficacy is limited as each replacement is bounded by the length of words.

*Deduplication* [3] is an existing redundancy elimination technique for file systems, which replaces repeated data chunks of a file with the matching chunks of other files in the system and is also used to reduce network traffic using concurrent data [17]. It is widely used and shown to be effective in large-scale storage systems and cloud storage services as their users are observed to store a wide variety of redundant files such as program packages and video files of high popularity. For instance, a cloud storage service run by Google [18] treats a file to upload to the cloud as uploaded instantly when the same file of the same hash value exists somewhere in the cloud storage. Dropbox is also known to use Deduplication in conjunction with delta encoding [19] that is another popular method for the version management of files as in *Diff*[20]. Because Deduplication mostly focuses on high volume data, it gives little attention to text-oriented data such as documents and webpages, which are relatively small. A popular open-source implementation of Deduplication, OpenDedup [21] also gives its minimum chunk size option for redundancy elimination, starting from 1 kB. 1kB is sufficiently small and detailed for large files but it is way too large for documents. Unless the documents of interest are exactly the same or are just different versions of the same file, redundancy elimination with 1kB chunk is not realistic. While OpenDedup only deduplicates exactly matching chunks, more recent Deduplication techniques such as [22] can find chunks with differences of a few bytes and are known to be more effective.

### 3 LZMA Primer

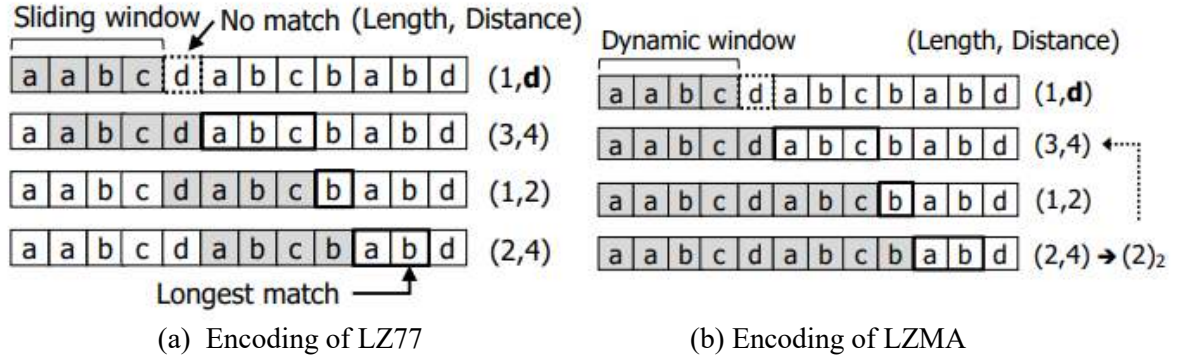


Fig. 1. Sample encoding of (a) LZ77 and (b) LZMA over a sequence of symbols. Whenever a match exists, the longest match is encoded with a length-distance pair. No match lets the symbol be encoded. When there is a distance value repeated recently, LZMA points to it instead of directly encoding it.

SynCoding is implemented based on LZMA. Therefore, in order to explain how SynCoding is implemented, we give a short primer of LZ77 and LZMA algorithms.

LZ77 encodes a sequence of symbols by maintaining a sliding window of size  $w$  within which the blocks of symbols appeared in the window are systematically constructed as a tree. Since the window is sliding, the blocks of symbols captured in the tree will change as the encoding proceeds. The compression of bits in LZ77 occurs when a repeated block of symbols is replaced with a length-distance pair, where the length and the distance denote the length of the block of symbols and the bit-wise distance from the current position to the position where the same block of symbols appeared earlier within the window. Every time a block of symbol is replaced by a length-distance pair, LZ77 tries to find the longest matching block in the window in order to reduce the number of encoded length-distance pairs as the reduction directly affects the compression efficiency. A sample encoding with LZ77 when the window size is 4 is illustrated in Fig. 1 (a). The static window size in LZ77 may cause inefficiencies. For example, when the window size is small, the number of blocks of symbols that can be kept in the window is limited, hence reducing the chances of compression.

LZMA works very similarly to LZ77 but with two major improvements. The first is that LZMA adopts a dynamic window that has its initial size as one and grows as the encoding proceeds. Because the window grows, LZMA is not suffering from being constrained by a small static window size. The second is that LZMA further reduces the number of bits representing a length-distance pair by specifying a few special encoded bits that are used when the current distance is the same with the distances that are most recently encoded. Reusing the distance information with fewer bits helps a lot when the data to compress has a repetitive nature (e.g., repetitive sentences or paragraphs in a file). The look up of the distances is typically done for the last four pairs. A sample encoding with LZMA is



depicted in Fig. 1 (b). These small changes cause LZMA can compress data more than LZ77 [23].

The optimality of LZ77 was proved earlier by Ziv and Lempel [24] in the sense that the total number of bits required to encode a data with LZ77 converges to the entropy rate of the data, where the entropy rate is defined with the symbol-by-symbol manner. Since LZMA is more efficient than LZ77, it is not difficult to prove that LZMA also converges to the entropy rate by extending the proof in [24].

Our interest lies whether SyncCoding uses less or more bits than LZMA. To this end, we explain how the number of bits required for LZMA can be mathematically evaluated.

Let  $T_{\text{LZMA}}(\{S\}_1^N)$  be the total required bits of the output encoded by LZMA for a given sequence of  $N$  symbols  $\{S\}_1^N$ . Suppose that  $p_{\text{LZMA}}$  is the number of phrases to be encoded in LZMA, where a phrase is defined by a block of symbols. Note that as the encoding progresses, the length of a new phrase (i.e., the number of symbols in the phrase) is determined by the longest matching sub-sequence of symbols that can be found in the sliding window. Then,  $T_{\text{LZMA}}(\{S\}_1^N)$  becomes the bits required to encode all the length-distance pairs for the phrases,  $\sum_{i=1}^{p_{\text{LZMA}}} \{f(l_i) + g(d_i)\}$ , where  $l_i$  is the length of phrase  $i$ ,  $d_i$  is the matching distance of phrase  $i$ , and  $f(l_i)$  and  $g(d_i)$  denote the bits to encode  $l_i$  and  $d_i$ , respectively. The matching distance  $d_i$  is the bit-wise distance from the current position to the previous position of the same phrase.

LZMA uses comma-free binary encoding [24] for  $f(l_i)$ , which is also used in LZ77. The comma-free binary encoding consists of two parts: 1) the prefix and 2) the binary encoding of  $l_i$ , denoted by  $b(l_i)$ . According to [24], the prefix and the binary encoding occupies  $2\lceil\log_2\lceil\log_2(l_i + 1)\rceil\rceil$  and  $\lceil\log_2(l_i + 1)\rceil$  bits, respectively. The summation of those quantifies  $f(l_i)$  of LZMA.

$g(d_i)$  in LZMA falls into either of the following three cases. When the distance to encode is not the same with any of the four recently used distances, the distance is encoded by the binary encoding of a fixed number of digits which is determined by the size of the sliding window  $w$ . Therefore  $g(d_i)$  always goes to  $\log_2 w$ . There is one exception when  $l_i = 1$  (i.e., the phrase consists of a single symbol), the symbol itself is encoded instead of the distance being encoded. Therefore,  $g(d_i) = \log_2 C$ , where  $C$  denotes the size of the symbol space (i.e., character space for a text encoding). When the distance is repeated from the four recently used distances, there exist two bit mappings of 4 bits or 5 bits by the following cases: 1)  $g(d_i) = 4$  when the distance matches with the first or the second lastly used distance, 2)  $g(d_i) = 5$  when the distance matches with the third or the fourth lastly used distance.

By the above equations, we can estimate the best case of LZMA, that happens when all the distances to encode for the phrases whose length is larger than two are found from the first or the second lastly used distance, i.e.,  $g(d_i) = 4$ . Thus, we have the following lower bound for  $T_{\text{LZMA}}(\{S\}_1^N)$ .

**Lemma 1**  $T_{\text{LZMA}}(\{S\}_1^N)$  is lower bounded by the following minimal possible total number of bits of LZMA:



$$T_{LZMA}(\{S\}_1^N) \geq p_{LZMA}^1 \cdot \lceil \log_2 C \rceil + 4(p_{LZMA} - p_{LZMA}^1) \\ + \sum_{i=1, |l_i| \neq 1}^{p_{LZMA}} (2\lceil \log_2 \lceil \log_2(l_i + 1) \rceil \rceil + \lceil \log_2(l_i + 1) \rceil),$$

where  $p_{LZMA}^1$  is the number of phrases whose length is one (i.e.,  $l_i = 1$ ).

## 4 System Design and Analysis

In this section, we formally state the problem that SyncCoding tackles and proposes the design of SyncCoding. Then, we provide a mathematical analysis for the design and explain how it can be compared with that of LZMA.

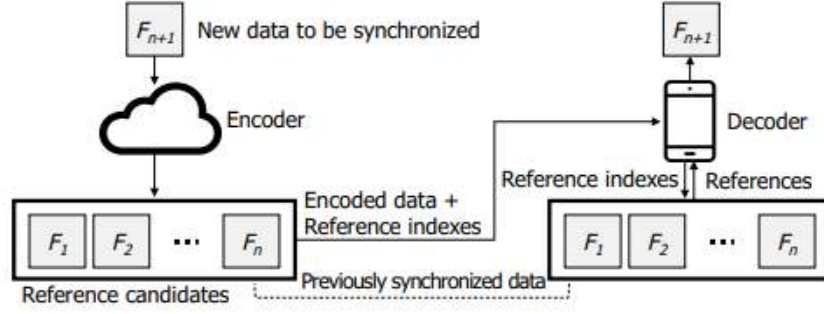


Fig. 2. The concept and basic operations of SyncCoding

### 4.1 System Design

Suppose that there exist  $n$  files that are previously synchronized between a server and a client, denoted by  $F_i$  where  $i = 1, \dots, n$ . Upon transmitting the  $(n + 1)$ -st file  $F_{n+1}$ , from the server to the client, our problem is to answer how should  $F_{n+1}$  be encoded using the shared files,  $F_1, \dots, F_n$ . Fig. 2 depicts this scenario in which the encoder (i.e., server) locates in a cloud system and the decoder is of a mobile device.

Given that the number of previously synchronized, we assume that we can somehow choose the most useful  $k$  files out of  $n$  files and use them only to encode  $F_{n+1}$ . We call those chosen files *references* and denote the set of references for  $F_{n+1}$  whose cardinality is  $k$  as  $R_{n+1}^k$ . Let us discuss the methods for choosing such  $k$  files in the next section.

For the compression, we let SyncCoding concatenate all the files in  $R_{n+1}^k$  to be a single large file and append it at the front part of  $F_{n+1}$  to create a virtual file to encode. We denote this virtual file, a compound of the file to encode and its references as  $V_{n+1}^k$ . Given  $V_{n+1}^k$ , we let SyncCoding simply encode it by LZMA in the hope that all the blocks of symbols that are commonly found in the references and the file to encode get converted to length-distance pairs, hence reducing the bits to encode. Note that when  $V_{n+1}^k$  is constructed, we let SyncCoding place the references in the order that a reference with higher usefulness is placed closer to  $F_{n+1}$ . Once encoding is done, we cut out the front part and extract only the encoded portion of  $F_{n+1}$ , denoted by  $E_{n+1}^k$ . SyncCoding transmits  $E_{n+1}^k$  to the decoder with the list of file indexes chosen as references, denoted by  $I_{n+1}^k$ .

For decoding  $E_{n+1}^k$ , we let SyncCoding first decode  $I_{n+1}^k$  to recall the references at the decoder side. Then, we let SyncCoding create the concatenated file of  $R_{n+1}^k$  as if it was done at the encoder and compress it by LZMA. Once we get the output, we append it at the front part of  $E_{n+1}^k$  to create a compound and decode the compound by LZMA. By the nature of LZMA, this decoding guarantees the acquisition of  $F_{n+1}$  from  $E_{n+1}^k$ . The encoding and decoding procedures of SyncCoding is summarized in **Algorithm 1**. We implement SyncCoding of this procedure by modifying an open-source implementation of LZMA [25].

---

**Algorithm 1** Encoding/Decoding Procedures of SyncCoding

---

**Encoding:**

- 1) Choose  $k$  useful references  $R_{n+1}^k$ , and index them by  $I_{n+1}^k$
- 2) Sort the references in  $R_{n+1}^k$  in the reverse order of usefulness
- 3) Concatenate all the references in  $R_{n+1}^k$
- 4) Append it at the front of  $F_{n+1}$  to get  $V_{n+1}^k$
- 5) Encode  $V_{n+1}^k$  by LZMA and cut out the encoded file  $E_{n+1}^k$
- 6) Transmit  $E_{n+1}^k$  and  $I_{n+1}^k$

**Decoding:**

- 1) From  $I_{n+1}^k$ , restore the concatenated file made up of  $R_{n+1}^k$
  - 2) Compress it by LZMA
  - 3) Append the compressed file at the front of  $E_{n+1}^k$
  - 4) Decode the compound by LZMA and cut out to obtain  $F_{n+1}$
- 

## 4.2 Comparative Analysis

We analyze SyncCoding by comparing its total number of bits for encoding,  $T_{\text{SC}}(\{S\}_1^N)$ , with that of LZMA. Recall that the input is again  $\{S\}_1^N$ , a sequence of  $N$  symbols, which was identically used for LZMA. By the analogy with the analysis of LZMA, we can view that  $T_{\text{SC}}(\{S\}_1^N)$  conforms to  $\sum_{i=1}^{p_{\text{SC}}} \{f(l_i) + g(d_i)\} + k \log_2 n$ , where  $p_{\text{SC}}$  denotes the number of phrases to be encoded in SyncCoding.  $k \log_2 n$ , the overhead of SyncCoding, quantifies the number of bits to list the indexes of the references. Since SyncCoding adopts LZMA for its bit encoding,  $f(\cdot)$  and  $g(\cdot)$  for SyncCoding are not different from those in LZMA. Note that the number of phrases identified in SyncCoding is always smaller than or at least equal to that in LZMA mainly because the references give a more

abundant source of matching phrases. Therefore, the better the reference selection, the more the gap between  $p_{SC}$  and  $p_{LZMA}$ . It is also obvious that  $p_{SC}^1 \leq p_{LZMA}^1$ , where  $p_{SC}^1$  denotes the number of phrases of length one in SyncCoding.

We now find the condition that guarantees better compression for SyncCoding over LZMA, so that  $T_{SC}(\{S\}_1^N) < T_{LZMA}(\{S\}_1^N)$  is satisfied. For that, we compare the worst case bit-size of SyncCoding with the best case bit-size of LZMA. Suppose that SyncCoding reduces the number of phrases by the factor of  $\gamma$  as  $p_{SC} = \gamma \cdot p_{LZMA}$ , where  $\gamma$  is a constant satisfying  $0 < \gamma \leq 1$ . It is unlikely, but if the reference selection goes extremely wrong, it is possible to have  $\gamma = 1$ . Having a smaller number of phrases that is to encode a smaller number of length-distance pairs is the key factor of reducing bits to encode for SyncCoding. However, this brings a side effect, which is to increase the average phrase length. Note that the ratio between numbers of phrases in LZMA and SyncCoding,  $\gamma$ , affects the average phrase length because the following holds:  $\bar{l}_{SC} \cdot p_{SC} = N$ , where  $\bar{l}_{SC}$  is the average phrase length in SyncCoding. Therefore, the average phrases length in SyncCoding increases by the factor of  $1/\gamma$  compared to LZMA as in  $\bar{l}_{SC} = \bar{l}_{LZMA}/\gamma$ , where  $\bar{l}_{LZMA}$  is the average phrase length in LZMA. Also, there is another side effect that is the increment in the distance of a length-distance pair. This increment may request more bits to encode the distance. The largest increment in bits comes from the case when a phrase finds its match from the farthest reference (i.e., the reference appended at the very beginning). Thus, this largest bit increment is affected by the number of references and is bounded by  $\log_2 k$  bits. Under this setting, we derive an upper bound of the bit-size of SyncCoding by assuming possible worst cases in combination as follows: 1) the distance to encode in each length-distance pair is either not found from any of the four lastly used distances or not of the length one, 2) the phrases to encode whose length is one are fully removed by using the references, say  $p_{SC}^1 = 0$ . The condition 1) makes each distance to be encoded by the binary encoding, so  $g(d_i) = \lceil \log_2 N \rceil$  holds. The condition 2) makes a phrase always encoded by a length-distance pair instead of being encoded by the symbol space, whose bit consumption  $\log_2 C$ , is typically much smaller than  $(d_i) = \lceil \log_2 N \rceil$ . These arguments with the Jensen's inequality<sup>2</sup> let us conclude that  $T_{SC}(\{S\}_1^N)$  is upper bounded by the following lemma.

**Lemma 2**  $T_{SC}(\{S\}_1^N)$  is upper bounded by the following maximal total number of bits:

$$\begin{aligned}
 T_{SC}(\{S\}_1^N) &\leq k \lceil \log_2 n \rceil + \gamma \cdot p_{LZMA} \cdot (\lceil \log_2 N \rceil + \lceil \log_2 k \rceil) \\
 &\quad + \gamma \cdot p_{LZMA} \cdot (2 \lceil \log_2 \lceil \log_2 (\bar{l}_{LZMA}/\gamma + 1) \rceil \rceil + \lceil \log_2 (\bar{l}_{LZMA}/\gamma + 1) \rceil).
 \end{aligned}$$

By using the Lemmas 1 and 2, the condition,  $T_{SC}(\{S\}_1^N) < T_{LZMA}(\{S\}_1^N)$ , gives the following theorem.

---

<sup>2</sup> For a random variable  $X$  and a concave function  $g$ ,  $E[g(X)] \leq g(E[X])$  holds. Such  $g$  includes  $\log_2 \cdot$  function.

**Theorem 1** *If  $h(\gamma) > 0$  is satisfied for the following definition of  $h(\gamma)$ , SyncCoding always compresses the same sequence of symbols more than LZMA.*

$$h(\gamma) = \alpha - \gamma \cdot p_{LZMA} \cdot (\beta + \lceil \log_2(\bar{l}_{LZMA}/\gamma + 1) \rceil + 2\lceil \log_2[\log_2(\bar{l}_{LZMA}/\gamma + 1)] \rceil), \quad (1)$$

where  $\alpha$  and  $\beta$  denote  $\sum_{i=1, |l_i| \neq 1}^{p_{LZMA}} (2\lceil \log_2[\log_2(l_i + 1)] \rceil + \lceil \log_2(l_i + 1) \rceil) + p_{LZMA}^1 \cdot \lceil \log_2 C \rceil + 4(p_{LZMA} - p_{LZMA}^1) - k\lceil \log_2 n \rceil$  and  $\lceil \log_2 N \rceil + \lceil \log_2 k \rceil$ , respectively.

It is complex to find the solution for  $\gamma$  that guarantees  $h(\gamma) > 0$ , but it is not difficult to show numerically that there exists  $\gamma < 1$  satisfying  $h(\gamma) > 0$ . Also, it is trivial that  $h(\gamma) > 0$  if  $\gamma$  approaches to zero. This implies that selecting references that effectively reduces the number of phrases to encode is the key for SyncCoding to be superior than LZMA.

### 4.3 Questions on SyncCoding

As revealed by the analysis, the efficacy of SyncCoding over LZMA depends highly on how much SyncCoding can reduce the number of length-distance pairs to encode. The ratio of reduction,  $\gamma$ , is the outcome of the reference selection. The question on which selection of a set of references from the synchronized data whose volume may be huge is the most efficient selection, brings the subsequent questions: 1) *which data in the synchronized data helps the most?*, 2) *what is the size of the set of references that leads to the best compression?*, and 3) *how long does it take for SyncCoding to encode and to decode a file with the chosen references (i.e., encoding and decoding complexity)?* 4) *How much energy does SyncCoding consume in downloading and decoding a file in mobile devices?*

It is essential to answer these questions to make SyncCoding viable in general data synchronization services, but answering each of these questions is challenging. Because of the complexity involved in the symbol tree construction in LZMA and also due to the correlated nature of symbols in the input sequence of symbols (e.g., language characteristics and intrinsic data correlation), none of the four questions can be tackled analytically. In the next section, we empirically characterize SyncCoding and give heuristic answers to these questions.

## 5 Characterization of SyncCoding

### 5.1 Reference Selection

We first tackle the question on reference selection. As it was intuitively explained in the system design, it is obvious that a file containing high similarity with the target file to encode is preferred to be included in the set of references. However, given that SyncCoding as well as LZMA tries to minimize the number of length-distance pairs to encode by seeking the longest matching subsequence of symbols, it is unclear how this similarity between files in the context of encoding can be defined. One definition rooted from the usefulness as a reference can be *the total length of matching subsequences included in the reference give a target file to encode*. The more the matching subsequences and the longer the matching subsequences, this definition gives a higher similarity value. However, this definition is practically impaired as its measurement itself takes as much time as the encoding process takes, so it is not so different from quantifying how much additional compression is obtained in SyncCoding by having the reference afterwards.

In order to ensure practicality, we need a much lighter similarity measure that can quickly investigate the individual usefulness of all the previously synchronized files with respect to the target file to encode. For this, we borrow the concept of document similarity, which has been widely used in the machine learning field with various implementations such as cosine similarity [26] and Kullback–Leibler divergence [27]. Based on such similarity measures, we propose a modified cosine similarity measure. Our modified cosine similarity denoted by  $\text{sim}(A, T)$  between two files, a reference candidate  $A$  and the target file to encode  $T$ , is formally defined as follows:

$$\text{sim}(A, T) \triangleq \frac{\sum_{i \in S(T)} f(t_i^A) f(t_i^T)}{\sqrt{\sum_{i \in S(T)} f(t_i^A)^2} \sqrt{\sum_{i \in S(T)} f(t_i^T)^2}}, \quad (2)$$

where  $S(T)$  is the set of distinct symbols  $\{t_i\}$ , observable from  $T$ ,  $t_i^A$  and  $t_i^T$  are the frequencies of observing the symbol  $t_i$  in the file  $A$  and  $T$ , and  $f(\cdot)$  is a transformation function. By definition,  $t_i^A \geq 0$  and  $t_i^T \geq 1$  hold.

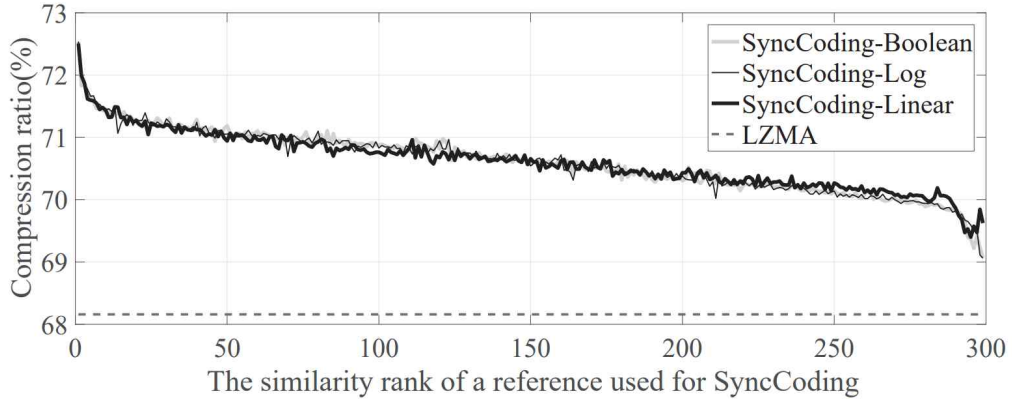


Fig. 3. The compression ratios of LZMA and SyncCoding with one reference whose modified cosine similarity is ranked by either of Boolean, Log, and Linear. Overall, SyncCoding with a single reference shows higher compression ratios than LZMA and a reference of a higher rank achieves a better compression ratio

In order to validate the efficacy of the proposed similarity measure, we randomly chose and downloaded a research paper from Google scholar [27] with a keyword “wireless networking”, which is arbitrarily chosen. To imitate the database of previously synchronized data for the chosen document, we have also downloaded three hundreds of research papers that came up with the same keyword as candidate references. With this sample data set, we rank the candidate references with the modified cosine similarity of three different  $f(\cdot)$  transformation functions for the chosen document: 1) Linear:  $f(t_i) = t_i$ , 2) Log:  $f(t_i) = \log(t_i + 1)$ , and 3) Boolean:  $f(t_i) = 1$  for  $t_i > 0$  and  $f(t_i) = 0$  for  $t_i = 0$ . We depict the compression ratio of SyncCoding with different  $f(\cdot)$  for each candidate reference sorted by its similarity rank in Fig. 3 in comparison with LZMA that uses no reference. Note that the compression ratio is the fraction of the compressed amount over the size of the original file, where the compressed amount is the difference between the size of the original file and the compressed file. Fig. 3 shows that SyncCoding with either of three functions compresses the chosen document more than LZMA. Especially with the reference candidate of the highest similarity rank, SyncCoding-Boolean achieves about 72.6% compression ratio meaning that the compressed size is only 27.4% of the original size. Comparing this result with that of LZMA which achieves the compression ratio of 68.2% and results in the compressed file whose size is 31.8% of the original, SyncCoding reduces the size of the compressed file by about 13.9% only with one well-chosen reference. Also, as Fig. 3 shows, SyncCoding with either of three functions maintains non-decreasing tendency over the reference candidates sorted by the rank. This implies that it is acceptable to use the modified cosine similarity rank for a quicker selection of a reference.

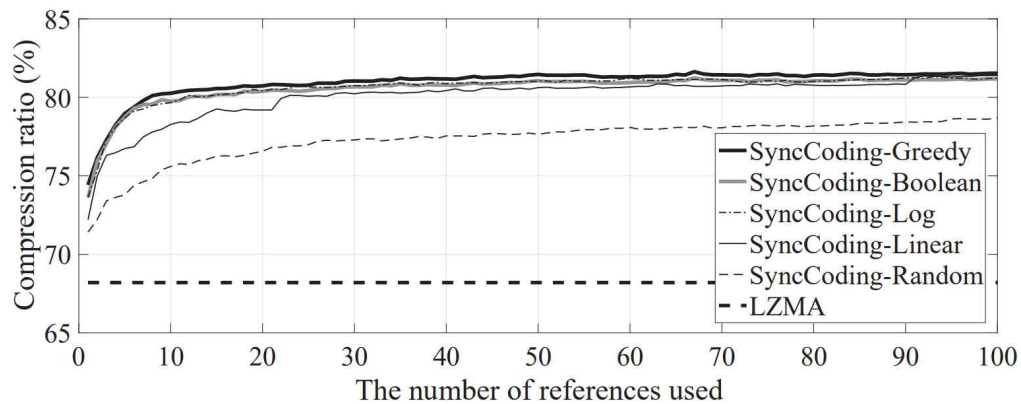


Fig. 4. The compression ratios of SyncCoding with increasing number of references that are selected

randomly, by a greedy search, or by the modified cosine similarity rank with either of Boolean, Log, or Linear. In this figure, the overhead for reference indexing is not considered to focus on understanding the impact of reference selection.

Fig. 4, where we increasingly add references for SyncCoding by the similarity rank measured by either of three functions, further investigates the efficacy of using the modified cosine similarity in the reference selection. In Fig. 4, we also include, for comparison, the compression ratios from a greedy search where the reference that maximally improves the compression ratio out of all remaining references is added to the existing set of references and from a random addition. Note that Fig. 4 only takes the size of the compressed amount into account when evaluating the compression ratio and does not consider the overhead of indexing the references, which will be discussed in the next subsection. As shown in Fig. 4, SyncCoding-Boolean performs better than others at least slightly and achieves the closest performance to the greedy search. Given that the computational complexities of the greedy search and SyncCoding-Boolean are  $O(N^2)$  and  $O(N)$ , respectively<sup>3</sup>, it is reasonable to conclude that SyncCoding-Boolean is a viable solution to the reference selection problem. Throughout this paper, we use SyncCoding-Boolean as our default SyncCoding implementation.

## 5.2 Maximum Compression Efficiency

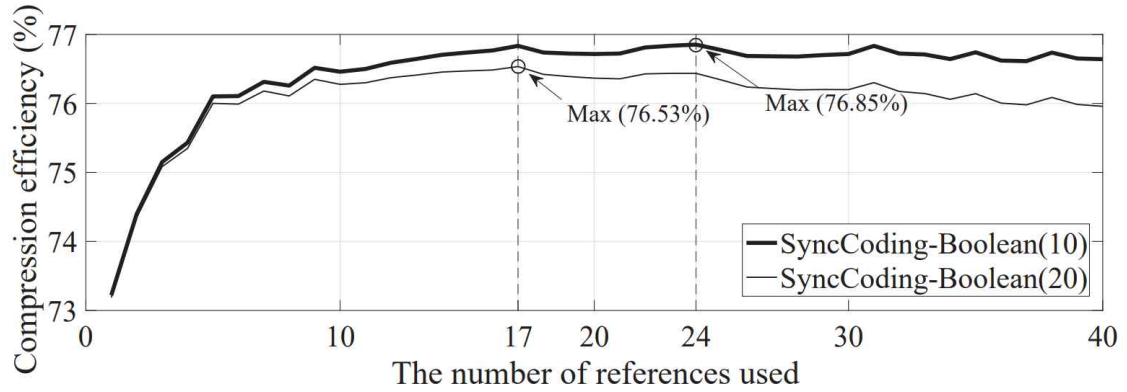


Fig. 5. The compression efficiency of SyncCoding for an increasing number of references. The per-reference overhead is chosen as either of 10 or 20 bytes.

<sup>3</sup> SyncCoding-Boolean incurs the complexity of evaluating  $N$  reference candidates linearly, where as the greedy search incurs the complexity of  $\sum_{j=N}^{N-k+1} j$  in order to find out the most helpful reference at every addition. The optimal can be obtained by a full search, but incurs  $O(N!)$ .



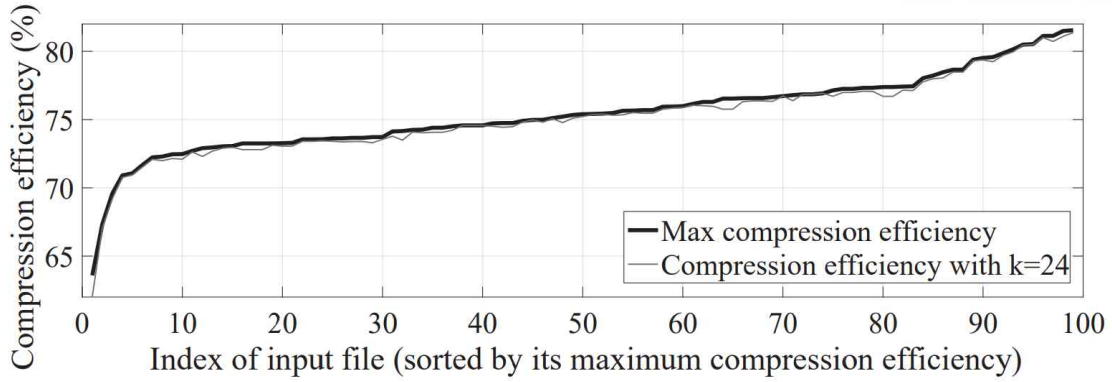


Fig. 6. Maximum compression efficiencies of SyncCoding obtained from 100 randomly chosen documents are compared with the compression efficiencies of SyncCoding that use only 24 references. Only a little gap exists.

We now tackle the second question on the maximum compression advantage of SyncCoding over LZMA. It is of particular interest in cases where the network bandwidth to deliver the compressed data is severely limited. The cases not only include extreme situations such as deep sea communication, inter-planet communication, but also include networks with high link cost such as satellite communication, while being at an ocean cruise or at an airplane. From a different perspective, it is also of strong interest in the cases where even a small amount of additional compression gives huge benefit. A nice example is found in inter-data center synchronization in which tens of terabytes are easily added daily and need to be synchronized (e.g., 24 terabytes of new videos are added to YouTube daily [29]).

If there is no overhead of listing the indexes of the references used for encoding, it is obvious that adding a new reference keeps improving the compression ratio of SyncCoding although the gain achieved by each addition may keep diminishing as shown in Fig. 4. However, SyncCoding requires the indexes to be independently encoded and transmitted along with the main data. We simply let SyncCoding use the address space of ten bytes, that is of 80 bits. This size of address (i.e., index size) gives a pointer that can specify a file from a database with about  $10^{24}$  files. It is relatively a large number for a personal use, but in the case of a global data center, it can be extended to twenty bytes (160 bits) or more to index the files with active accesses. To characterize the impact of the overhead from the indexes in SyncCoding, we depict SyncCoding-Boolean with two index sizes, considering the overhead added to the size of the compressed file in Fig. 5. To avoid confusion, we define *compression efficiency* as the compression ratio evaluated with the compressed amount including the overhead, i.e., the ratio between the compressed amount plus overhead and the original file size. For simplicity, we quantify the overhead by the address space size multiplied with the number of references used, meaning that no additional encoding is applied for the indexes. As shown in Fig. 5,

with 10 and 20 bytes overhead per reference, SyncCoding achieves about 76.85% and 76.53% as its maximum compression efficiency for the chosen document, respectively. The number of references that achieves the maximum compression efficiency is 24 and 17, confirming the intuition that a larger per-reference overhead makes the compression efficiency saturated earlier with respect to the number of references used. However, even with a larger per-reference overhead, the maximum compression efficiency achieved does not change much. This is because the referencing happens mostly from a small number of highly similar files. With 10 bytes per-reference overhead, we further obtain the results about the maximum compression efficiency by repeatedly applying SyncCoding to a randomly selected document from our research paper dataset in Section 5.1 with all the unselected documents used as reference candidates for one hundred times. In Fig. 6, we depict the maximum compression efficiency from 100 input documents in the ascending order and for the same input document we also plot the compression efficiency achieved with 24 references, the optimal number of references in Fig. 5. As shown in Fig. 6, SyncCoding works well enough with only 24 references and the gain of using more references is less than 1.5%. Therefore, unless specified otherwise, we opt to use  $k^* = 24$  as our default number of references for all the following experiments.

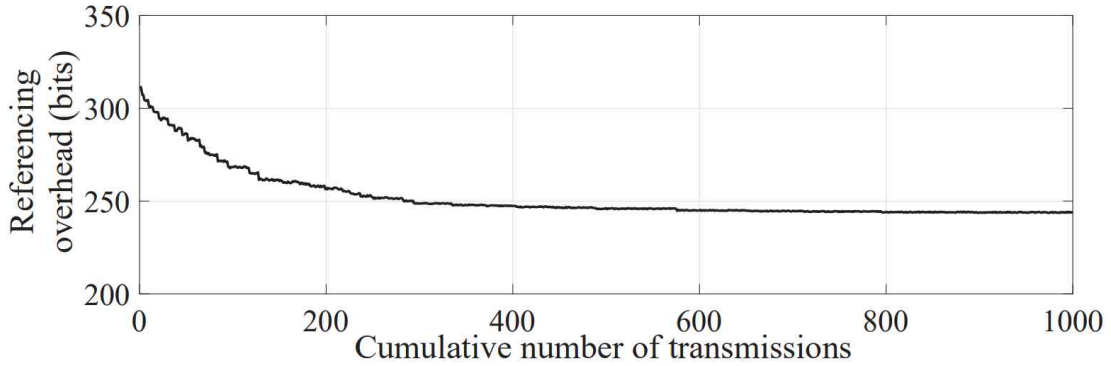


Fig. 7. The referencing overhead by indexing references with Huffman coding when referencing frequencies of reference candidates are updated over transmissions.

### 5.3 Referencing Overhead Optimization

The overhead of referencing files by fixed-length indexes can be further optimized by a variable length coding such as Huffman coding [10]. Especially when there are multiple Fig. 6. Maximum compression efficiencies of SyncCoding obtained from 100 files to exchange between end hosts which already have many synchronized files, for instance file exchange between data centers, indexing with a variable length coding can help. In such a case, instead of indexing  $N$  files equally assigned with  $\log_2 N$  bits, assigning less bits for more frequently referenced files is possible. Huffman

coding in principle assigns  $-\log_2 w_i$  bits to index file  $i$  that is referenced by  $f_i$  times thus having its relative weight  $w_i = f_i / \sum_{j \in [N]} f_j$ . Similarly, at every file transfer between end hosts,  $w_i$  can be updated for all synchronized files and the referencing indexes can be reassigned accordingly. The more the files are referenced, the less the bits are reassigned. Hence, referencing overhead reduces as file transfers continue.

In order to demonstrate this idea, we collect 8,000 RFC documents from [30] and build a synchronization database between end hosts, in which indexing a reference needs 13 bits for equal bit assignment. We test the total referencing overhead for each file transfer where a thousand randomly chosen files from the database are transferred sequentially each with 24 references, and the indexes are updated as aforementioned. In a test, all other unchosen files are considered previously synchronized and are used as reference candidates. The results in Fig. 7 are averages from 200 repetitions of this test. As shown in the figure, the overhead starts from 312 bits ( $13 \times 24$ ) and reduces gradually over transmissions to about 245 bits. It is hard to know what the actual reduction in overhead will be because it depends on how frequencies of chosen references change over transmissions, but it is always possible to optimize overhead in this way. In particular, data centers with a huge number of synchronized files can benefit from this more. However, in the remaining sections, we use fixed-length indexes to characterize the performance of SyncCoding the most conservatively.

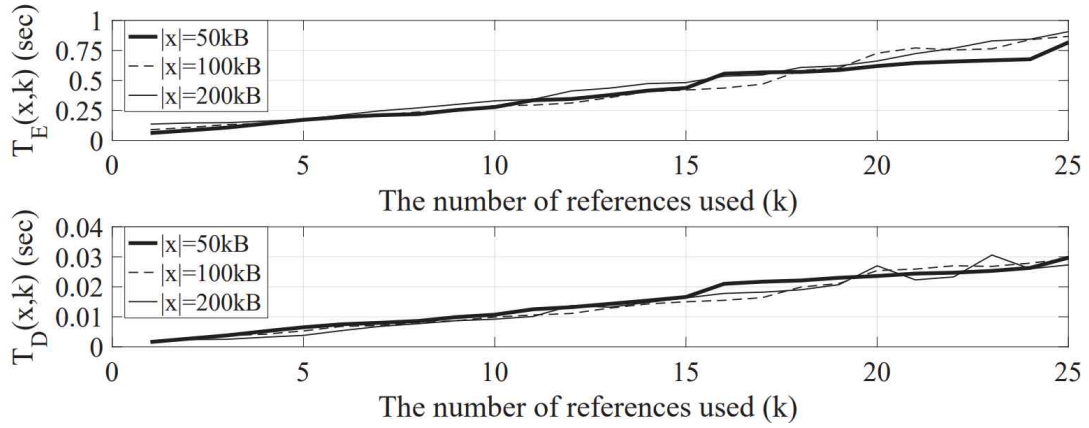


Fig. 8. Experimental evaluation of  $T_E(x, k)$  (top) and  $T_D(x, k)$  (bottom), the time durations to encode and to decode a file  $x$  with  $k$  references under Intel i7-3770 CPU (3.40 GHz).

#### 5.4 Encoding Time and Decoding Time of SyncCoding

We tackle the third question on the encoding and the decoding time of SyncCoding by performing experiments. We let  $T_E(x, k)$  and  $T_D(x, k)$  denote the time durations taken for encoding and

decoding a file  $x$  with  $k$  references. Because the encoding and the decoding complexities of SyncCoding with  $k$  references are not largely different from the complexity of LZMA repeated by  $k$  times, it is expected that  $T_E(x, k)$  and  $T_D(x, k)$  may increase linearly as  $k$  increases for a given  $x$ . Fig. 8, a measurement on Linux (Kernel 2.6.18-238.el5) over Intel i7-3770 CPU (3.40 GHz) for three kinds of research papers of about 50, 100, and 200 kB, randomly chosen from the aforementioned dataset in Section 5.1, confirms that the average encoding time as well as the average decoding time from one hundred trials increases almost linearly to  $k$ . Fig. 8 also confirms that the size of  $x$  has little impact on the times because the size of the data to encode is relatively smaller than the total size of the references. The decoding time is on the scale of milliseconds and is relatively negligible compared to the encoding time which is on the scale of a second. One important thing to note here is that the encoding time can often be hidden to users due to the following reasons: 1) the existence of a powerful encoding server, 2) the parallelism between the encoding process and the network transmission process, and 3) preprocessing of SyncCoding in the server. We will explain more about the applicability of the preprocessing of SyncCoding to practical use cases in Section 6.

### 5.5 Mobile Energy Consumption of SyncCoding

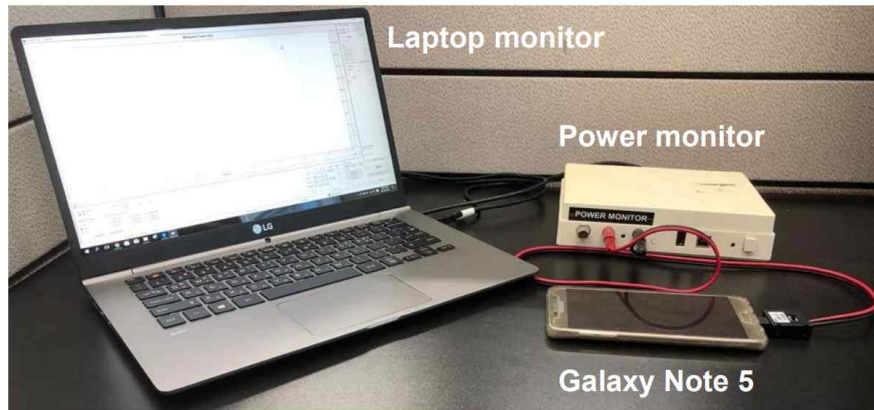
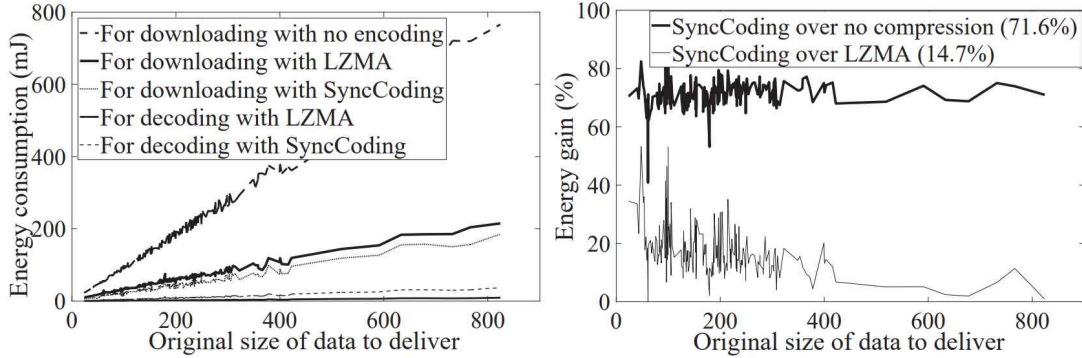


Fig. 9. Our test environment for measuring energy consumption of receiving data with or without SyncCoding in an Android device, Galaxy Note 5. Measurements are conducted by a digital power monitor from Monsoon [31].

We answer the last question on the mobile energy consumption of SyncCoding in this subsection. Since mobile devices are more prone to high energy consumption than powercorded desktops or data centers, it is important to know how much energy that SyncCoding consumes for receiving data in mobile devices. We define the energy consumption with a compression algorithm for receiving data to be the total energy consumption from the start of downloading data to the end of decoding data. When

no compression is applied, no energy is consumed for decoding. We experiment the energy consumption from SyncCoding in comparison with LZMA and no compression in two perspectives: 1) energy saving by downloading compressed data of smaller sizes and 2) extra energy spending for decoding. For this experiment, we reuse the research papers of the aforementioned dataset in Section 5.1, whose average file size is about 200 Kbytes.



(a) The energy consumption for downloading and decoding data of variable sizes with SyncCoding and LZMA.

(b) The energy gain of SyncCoding over no compression and LZMA. Average gains are presented in the bracket.

Fig. 10. The energy consumption measurement results on Galaxy Note 5 smartphone for downloading and decoding data of variable sizes with SyncCoding and LZMA.

We randomly choose one input file (i.e., the target paper to compress) from this dataset and use 24 reference files (i.e.,  $k^* = 24$ ) chosen out of all other papers. The experiment is carried out on an Android device, Galaxy Note 5 connected to an LTE network. The average downlink speed of the LTE network we use during the experiment is about 30 Mbps. Our setup for the energy measurement is depicted in Fig. 9.

Fig. 10 (a) shows the mobile energy consumption with SyncCoding, LZMA, and with no compression for receiving data of variable sizes. When receiving data with a compression method, the downloading size is reduced, but the additional decoding process is needed. SyncCoding on average saves 26.2% and 67.9% energy than LZMA and no compression for the downloading part but overspends 75.0% than LZMA the decoding part. Fig. 10 (b) summarizes the energy gain of SyncCoding over LZMA and no compression, which are shown to be on average 14.7% and 66.7%, respectively<sup>4</sup>. The energy consumption for downloading as well as decoding increases nearly linearly to the input file sizes. This is reasonable because the compressed data size is highly correlated with the input file size and the decoding process that reads through the compressed data to progressively recover the original bit sequences from the references is also highly correlated with the compressed

data size. Since the amount of saved energy in downloading is greater than the energy overspent for decoding, SyncCoding overall outperforms no compression as well as LZMA in terms of total mobile energy consumption.

We note that the energy gain of SyncCoding can be affected by network conditions or computational efficiencies of mobile devices. For example, if the downlink speed of the LTE network is much faster than 30 Mbps, then the consumed energy for downloading can be smaller than before due to the reduced downloading time. Also, if the computing efficiency of a device is worse than Galaxy Note 5 (e.g., Galaxy S2), then the consumed energy for decoding would be larger. Therefore, one should consider these conditions when applying SyncCoding for energy saving of mobile devices.



## 6 Evaluation

We evaluate the efficacy of SyncCoding in two real data synchronization services: 1) cloud data

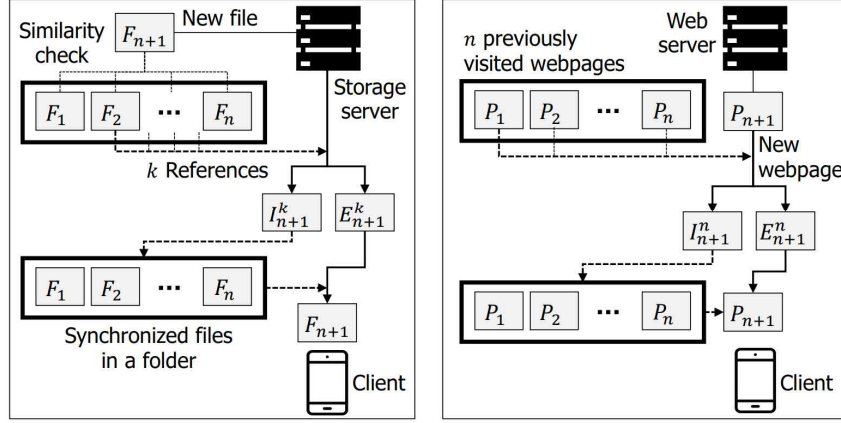


Fig. 11. Overview of the evaluation scenarios: 1) cloud data sharing (left) and 2) web browsing (right).

We evaluate the efficacy of SyncCoding in two real data synchronization services: 1) cloud data sharing and 2) web browsing. The scenario we consider for cloud data sharing is to synchronize a new file of an existing folder from the storage server to a user device, given that the folder already includes about a hundred files relevant to the new file. The use case we consider for web browsing is to browse webpages of a website at a user device given that the webpages visited up to a moment are all cached in the device, so the web server can exploit those cached pages for encoding a new page. The overview of these scenarios is depicted in Fig. 11.

We experiment both scenarios and statistically compare the compression efficiency of SyncCoding with existing encoding techniques, Brotli, Deflate, LZMA, and Deduplication, whose settings are described in the next subsection. Here we focus on the compression efficiency without being concerned about the encoding and the decoding time, in order to give our focus to the reduction of network data traffic. As is discussed in the previous section, applying SyncCoding on the fly takes time. Therefore, SyncCoding may not be effective in speeding up web browsing experiences especially for web servers with insufficient computational capability. However, SyncCoding is still useful to the users who would like to browse web pages with minimal cellular data cost. In the case of cloud data sharing where the users are less sensitive to the synchronization delay, the processing time for the SyncCoding can be successfully hidden to its users.

### 6.1 Settings

**SyncCoding:** We use SyncCoding-Boolean with  $k^* = 24$  references unless it is specified and use the

per-index overhead of 10 bytes. For the parameters inherited from LZMA implementation, we adopt the values from LZMA with its maximum compression option.

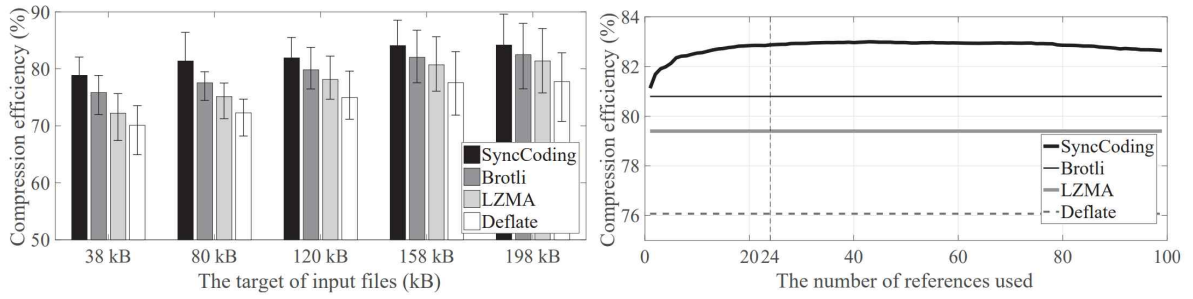
**LZMA:** For the evaluation of LZMA, we use its SDK (Software Development Kit) provided in [25] with the parameters from the maximum compression option.

**Deflate:** For the evaluation of Deflate, we use [32], a popular open source library including Deflate with all the parameters from the maximum compression option.

**Brotli:** For Brotli, we use an open source implementation of Brotli [33], which is embedded in Google Chrome web browser [34]. We also use its maximum compression option.

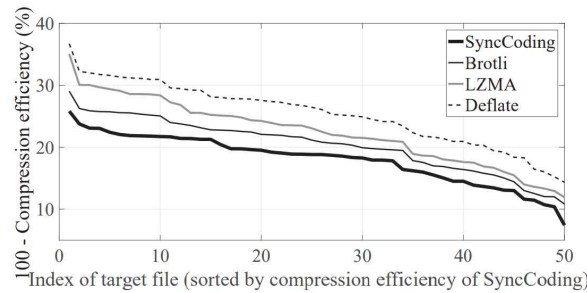
**Deduplication:** For the evaluation of Deduplication, we modify OpenDedup [21] so as to investigate its ideal Deduplication performance for documents. We reduce the lower bound of the chunk size (i.e., 1kB in OpenDedup) to be arbitrarily small.

## 6.2 Use Case 1: Cloud Data Sharing



(a) Compression efficiency for a target document of various sizes. The error bars indicate 90% confidence intervals.

(b) Compression efficiency for various numbers of references.



(c) Compressed size comparison for 50 target documents sorted by the value of SyncCoding.



Fig. 12. Compression efficiencies of SyncCoding and other techniques (a) for various document sizes to encode, (b) for various numbers of references. (c) A comparison of compressed sizes of 50 target documents when  $k^*$  references are used.

We emulate a folder of a cloud storage (e.g., Dropbox) by creating a folder with files of similar attributes. To fill the folder, we randomly downloaded two hundred RFC documents from [30], which are all in the format of TXT.

For the evaluation of SyncCoding and other encoding techniques except Deduplication, we regard a randomly chosen file from the folder as the target file to encode for synchronization and assume that all other files in the folder are reference candidates. We perform the following three tests and evaluate the compression efficiencies of SyncCoding and other techniques: 1) Tests for the target documents of various sizes with  $k^*$  references, 2) Tests for a randomly chosen document with various numbers of references, 3) Tests for 50 randomly chosen target documents with  $k^*$  references. In test 1), for each size of the target document, we select and test 20 documents whose size ranges from 90% to 110% of the given size. Fig. 12 summarizes the results of these tests. Fig. 12 (a) shows the average compression efficiencies with 90% confidence intervals for different sizes of documents to encode and reveals that SyncCoding persistently outperforms others. With respect to the compressed size (i.e., 100% - compression efficiency), SyncCoding makes the size on average 12.4%, 20.1%, and 29.9% less than Brotli, LZMA, and Deflate. Fig. 12 (b) shows that SyncCoding achieves nearly the maximum compression efficiency at around  $k^* = 24$  number of references, which was our rule of thumb for practical use. Fig. 12 (c) comparing the compressed sizes of 50 randomly chosen documents confirms that SyncCoding gives consistent saving over Brotli, LZMA, and Deflate of about 11.2%, 17.9%, and 29.2%.

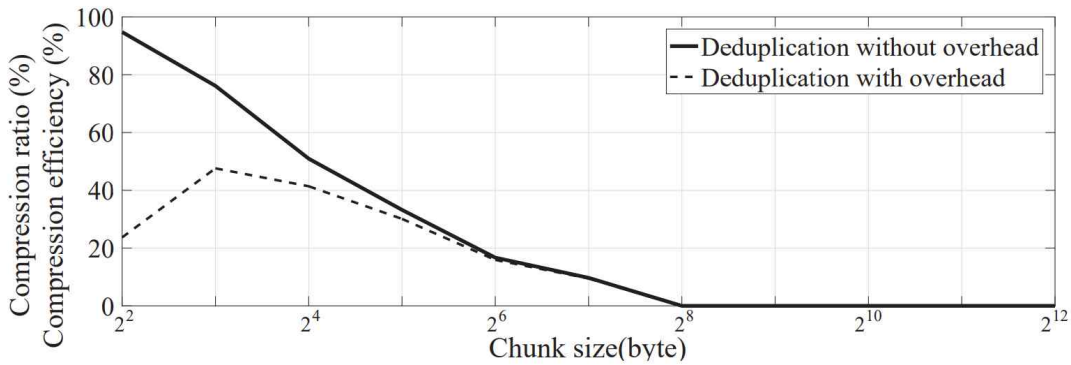


Fig. 13. Compression ratio and compression efficiency of Deduplication without overhead and with overhead for various chunk sizes in the cloud data sharing scenario with 100 reference files.

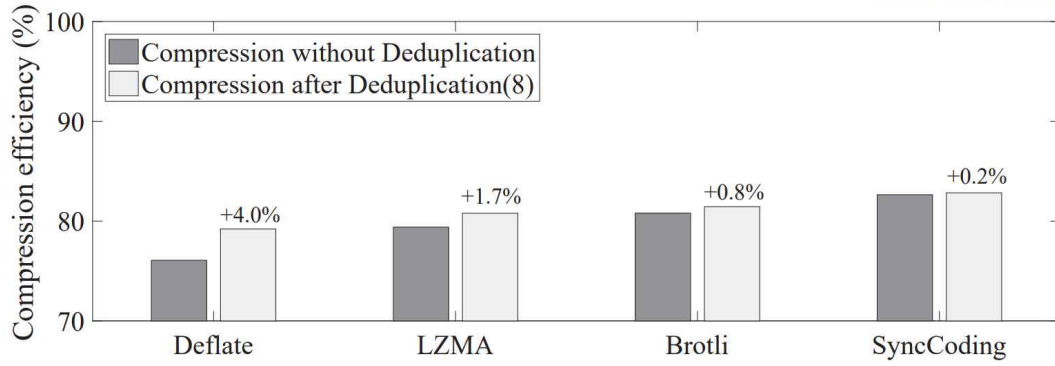
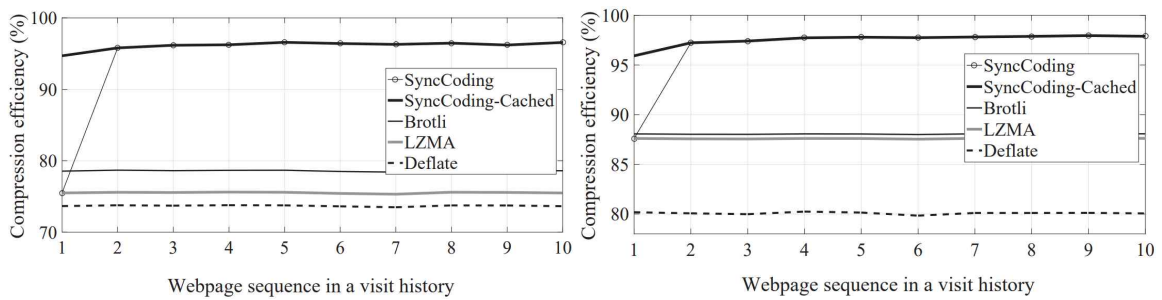


Fig. 14. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes.

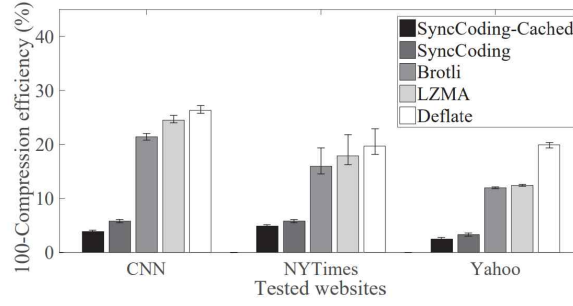
We separately test the performance of Deduplication from a randomly chosen target file with one hundred reference files for various chunk sizes from 4 to 4096 bytes. Fig. 13 shows the compression ratio and efficiency with and without overhead. As Fig. 13 shows, Deduplication achieves its maximum of about 47.60% when the chunk size is 8 bytes, but this is far lower than 82.65% from SyncCoding.

We further test the compression efficiencies of SyncCoding and other techniques over the outcomes of Deduplication with one hundred reference files and its best chunk size in Fig. 14. This mimics a mixed Deduplication and compression method proposed in [35]. Our experiment verifies that Deduplication indeed helps other encoding techniques by about 2.17% on average but helps SyncCoding by only about 0.22%. This limited improvement over SyncCoding implies that SyncCoding already eliminates most redundancy that Deduplication targets to eliminate.



(a) Sample compression efficiencies for the webpages in a visit history of CNN.

(b) Sample compression efficiencies for the webpages in a visit history of Yahoo.



(c) Average compressed sizes of the webpages from CNN, NYTimes, and Yahoo. The error bars indicate 95% confidence intervals.

Fig. 15. Compression efficiencies of SyncCoding, SyncCoding-Cached and three other encoding techniques for the webpages sequentially visited by sample visit histories obtained from (a) CNN (Politics section) and (b) Yahoo (Science section). (c) A comparison of the average compressed sizes of webpages from three websites with no section restriction.

### 6.3 Use Case 2: Web Browsing

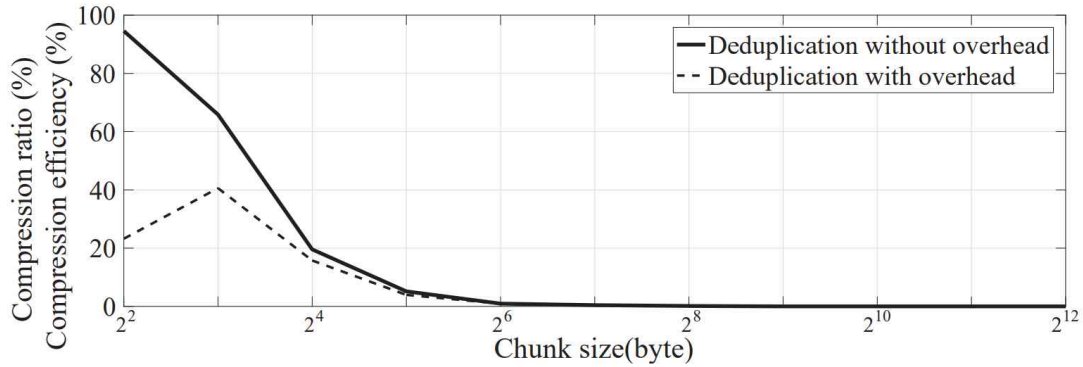


Fig. 16. Compression ratio and compression efficiency of Deduplication without and with overhead for various chunk sizes on a CNN webpage with 10 reference pages.

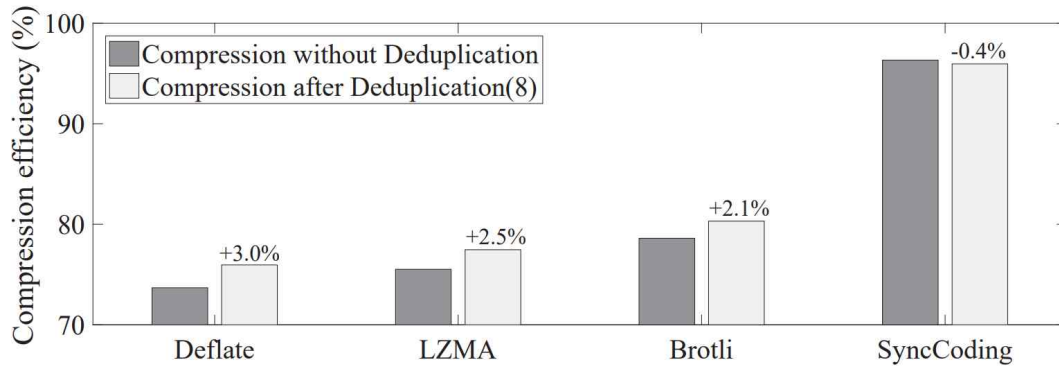


Fig. 17. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes.

To evaluate the efficacy of SyncCoding in web browsing, for a given website, we recorded webpage visit histories of a user and cached all the resources relevant to the webpages (e.g., HTML files, Java scripts, and CSS files) in the visit histories by an off-the-shelf web browser, Google Chrome.

For a given sequence of webpages in a history, we let encoding techniques in comparison compress each webpage when it is invoked. SyncCoding and Deduplication are assumed to utilize all the previous webpages to the newly invoked webpage and Brotli is assumed to exploit its pre-defined static dictionary, that is delivered in advance, between the server and the client. LZMA and Deflate do not use additional resources.

Fig. 15 (a) and (b) show the compression efficiency comparison for a sample visit history recorded inside the politics category of CNN and inside the science category of Yahoo. As expected, Fig. 15 (a) shows that SyncCoding does not show any advantage over LZMA when there is no previous webpage to use, i.e., for the first webpage. However, from the second webpage onward, SyncCoding shows significant compression efficiency improvement over LZMA, Brotli, and Deflate. The compression efficiency is nearly maximized after the third webpage and the improvement over Brotli is as much as 20% on average. The same pattern for the compression efficiency is observed for the webpages of Yahoo as shown in Fig. 15 (b). One important thing to note here is that if we allow SyncCoding to cache an old webpage of a website, for instance the main webpage of CNN or Yahoo of yesterday, to our surprise SyncCoding achieves from the first page as good compression efficiency as visiting the second page as shown in Fig. 15 (a) and (b). We denote this technique by *SyncCoding-Cached*. We wondered why this huge gain appears in SyncCoding and found the following reason by an analysis for the contents of the webpages: every webpage in a website authored by a company or a group of programmers show extremely similar programming style (e.g., programming templates), and thus a huge portion of the contents can be referenced from previous webpages in SyncCoding. Note that this gain is fundamentally not achievable when using a static pre-defined dictionary such as in Brotli. To evaluate the performance of SyncCoding for more general web browsing behaviors, we let two test users freely visit webpages of three websites for an hour, CNN, NYTimes, and Yahoo. Using their visit histories, we perform the same test and depict the average compression efficiencies with 95% confidence intervals in Fig. 15 (c). The figure confirms that in the perspective of the compressed size, the improvement of SyncCoding-Cached over Brotli, LZMA, and Deflate are on average 78.3%, 79.6%, and 86.1% even under such general browsing behaviors. This implies that if a website is prepared to serve its webpages with SyncCoding, it can substantially enhance its user experience.

We again evaluate the performance of Deduplication on the CNN case with ten reference pages for various chunk sizes. Fig. 16 shows the compression ratio and efficiency with and without referencing overhead. Fig. 16 shows that Deduplication achieves its maximum compression efficiency of about 40.59% when the chunk size is 8 bytes, but this is still far below 96.56% from SyncCoding.

We also test the compression efficiencies of SyncCoding and other techniques with ten reference pages and its best chunk size in Fig. 17. It shows Deduplication helps other encoding techniques by about 2.53% on average but makes SyncCoding even worse by about 0.4%. This is because SyncCoding loses some chances to match longer subsequences after the Deduplication which twists those subsequences as mixtures of original contents and the addresses to matching chunks.

## 7 Discussion

In this section, we discuss the performance issue of SyncCoding when it is applied to encrypted data. Several user applications including cloud storage services such as Dropbox or Google Drive transmit the data after encrypting it due to the security and privacy concerns. A natural question that arises is whether SyncCoding can compress even the encrypted data more or not? If the repeated patterns of data inherent in the file before it is being encrypted can be preserved in the encrypted file, SyncCoding may still be possible to compress the encrypted data more compared to other compression algorithms. In such a case, SyncCoding encoder and decoder can be implemented in network proxies located in edge servers and can improve the efficiency of data transmission without having any modification in the existing data synchronization applications.

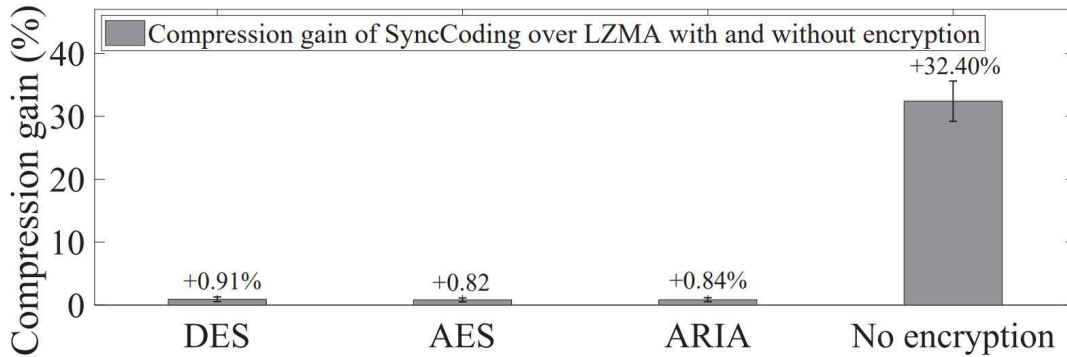


Fig. 18. The compression gain of SyncCoding over LZMA with 90% confidence intervals for three encryption algorithms and for no data encryption.

To evaluate the efficacy of SyncCoding over encrypted data, we test three popular encryption algorithms, DES, AES, and ARIA, explained as follows. DES (Data Encryption Standard) [36], [37] is a symmetric encryption algorithm whose encryption and decryption keys are the same, which had been used from 1975 as an encryption standard in the US, but it is relatively vulnerable due to its small key size of 56 bits. In 2001, AES (Advanced Encryption Standard) [38], [39] was developed by NIST (National Institute of Standard and Technology). It is also an encryption standard in the US which has been most commonly used for applications such as Dropbox and Google Drive. It uses variable key sizes from 128 to 256 bits and uses Rijndael algorithm [40], which uses substitution and permutation in each block encryption round. ARIA (Academy Research Institute Agency) [41], [42] is a block encryption algorithm that also uses variable key size like AES and it is an encryption standard in South Korea. Both AES and ARIA use symmetric keys. Here, we focus only on symmetric encryption algorithms since asymmetric encryption algorithms whose encryption and decryption keys are different, are not widely used for large data transmission over the Internet due to their much slower

encryption and decryption performance.

We reuse the same dataset as in the cloud data sharing scenario in Section 6.2. For the evaluation of SyncCoding, we choose one target document from the dataset and used  $k^* = 24$  references from all other documents.

Fig. 18 shows the average compression gain of SyncCoding over LZMA with 90% confidence intervals for three encryption algorithms when the data is compressed after and before encryption. We repeat each scenario 100 times and randomly choose target documents for each simulation. When the data is compressed after encryption, the gain of SyncCoding over LZMA for DES, AES, and ARIA in the compressed size are about 0.91%, 0.82%, and 0.84%, respectively. When the data is compressed before encryption, the gain of SyncCoding over LZMA is about 31.2%. SyncCoding shows little performance gain over LZMA for encrypted data. More interestingly, both algorithms merely reduce the size of the encrypted data. The compression efficiencies of SyncCoding for DES, AES, and ARIA are about 1.3%, 0.82%, and 0.71%, respectively. This is because of the features of modern encryption techniques: *Confusion* and *Diffusion*. *Confusion* makes it more difficult to guess the contents of the original data. *Diffusion* makes it harder to find the pattern of the encrypted data. By doing so, they transform original data into high-entropy data. Since the repeated patterns of original data are well hidden inside the encrypted data, compression over encryption is not effective. This implies that SyncCoding should be applied before encryption to obtain its benefit. In other words, SyncCoding should be embedded within the networking applications (i.e., server and client applications) and the encryption should be applied to data after SyncCoding makes it compressed.

## 8 Concluding Remarks

In this work, we propose a novel data encoding technique SyncCoding that exploits the database of previously synchronized data to improve efficiency of networking. Our experiments show that SyncCoding reduces the energy consumption of mobile devices in data download and show that SyncCoding outperforms existing encoding techniques, Brotli, Deflate, and LZMA in terms of compression efficiency in two popular use cases: cloud data sharing and web browsing. SyncCoding sets up a new baseline for encoding techniques that exploit inter-file correlations.



## REFERENCES

- [1] W. Nam, J. Lee, and K. Lee, "Synccoding: A compression technique exploiting references for data synchronization services," in *IEEE 25th International Conference on Network Protocols (ICNP)*, 2017.
- [2] Z. Bar-Yossef, Y. Birk, T. Jayram, and T. Kol, "Index coding with side information," *IEEE Transactions on Information Theory*, vol. 57, no. 3, pp. 1479–1494, 2011.
- [3] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage*, vol. 7, no. 4, p. 14, 2012.
- [4] N. Ranganathan and S. Henriques, "High-speed VLSI designs for Lempel-Ziv-based data compression," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, no. 2, pp. 96–106, 1993.
- [5] I. Pavlov. 7z format. <http://www.7-zip.org/7z.html>.
- [6] M. Mahoney, "Fast text compression with neural networks," in *AAAI*, 2000.
- [7] J. Alakuijala and Z. Szabadka, "Brotli compressed data format," Tech. Rep., 2016.
- [8] P. Deutsch, "DEFLATE compressed data format specification version 1.3," Tech. Rep., 1996.
- [9] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.
- [10] D. A. Huffman *et al.*, "A method for the construction of minimum redundancy codes," *Proceedings of the I.R.E.*, vol. 40, no. 9, pp. 1098–1101, 1952.
- [11] J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM Journal of research and development*, vol. 20, no. 3, pp. 198–203, 1976.
- [12] M. Ruhl and H. Hartenstein, "Optimal fractal coding is np-hard," in *IEEE Data Compression Conference (DCC)*, 1997.
- [13] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [14] R. Franceschini, H. Kruse, N. Zhang, R. Iqbal, and A. Mukherjee, "Lossless, reversible transformations that improve text compression ratios," *Technical Report, University of Central Florida*, pp. 1–33, 2000.
- [15] F. S. Awan and A. Mukherjee, "LIPT: A lossless text transform to improve compression," in *IEEE International Conference on Information Technology: Coding and Computing*, 2001.
- [16] J. Alakuijala and Z. Szabadka. (2014) IETF Brotli compressed data format. <https://tools.ietf.org/html/draft-alakuijala-brotli-01>.

- [17] Understanding Data Deduplication. <https://www.druva.com/blog/understanding-data-deduplication/>.
- [18] Improving the deduplication flow when uploading to Google Drive. <https://gsuiteupdates.googleblog.com/2016/09/improving-deduplication-flow-when.html>.
- [19] J. Mogul, B. Krishnamurthy, F. Douglass, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein, "Delta encoding in HTTP," Tech. Rep., 2001.
- [20] File version history–Dropbox. <https://www.dropbox.com/help/security/version-history-overview>.
- [21] Openedup – opensource dedupe to cloud and local storage. <http://openedup.org/odd/>.
- [22] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," in *ACM International Conference on Mobile Computing and Networking*, 2015, pp. 592–603.
- [23] Z. Tu and S. Zhang, "A novel implementation of jpeg 2000 lossless coding based on lzma," in *IEEE International Conference on Computer and Information Technology*, 2006.
- [24] A. D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 872–877, 1994.
- [25] I. Pavlov. Lzma sdk. <http://www.7-zip.org/sdk.html>.
- [26] N. Dehak, R. Dehak, J. Glass, D. Reynolds, and P. Kenny, "Cosine similarity scoring without score normalization techniques," in *Odyssey: The Speaker and Language Recognition Workshop*, 2010.
- [27] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.
- [28] Google scholar. <http://scholar.google.com/>.
- [29] R. Turek. What YouTube Looks Like In A Day [Infographic]. <https://beat.pexe.so/what-youtube-looks-like-in-a-dayinfographic-d23f8156e599#.wbu0v1h2z>.
- [30] IETF RFC Index. <https://www.ietf.org/rfc.html>.
- [31] "Monsoon solutions," <http://msoon.github.io/powermonitor/>.
- [32] J.-l. Gailly and M. Adler. zlib compression library. <http://www.zlib.net>.
- [33] Brotli compression format. <https://github.com/google/brotli>.
- [34] Google to boost compression performance. <http://www.computerworld.com/article/3025456/web-browsers/google-to-boost-compression-performance-in-chrome-49.html>.
- [35] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," in *Data Compression Conference (DCC), 2011*, 2011, pp. 393–402.

- [36] D. E. Standard, “Data encryption standard,” *Federal Information Processing Standards Publication*, 1999.
- [37] “Des source code (3-des / triple des) - mbed tls (previously polarssl),” <https://tls.mbed.org/des-source-code>.
- [38] F. P. Miller, A. F. Vandome, and J. McBrewster, “Advanced encryption standard,” 2009.
- [39] “Aes source code (advanced encryption standard) - mbed tls (previously polarssl),” <https://tls.mbed.org/aes-source-code>.
- [40] T. Jamil, “The rijndael algorithm,” *IEEE potentials*, vol. 23, no. 2, pp. 36–38, 2004.
- [41] “Aria block encryption algorithm - kisa,” <https://seed.kisa.or.kr/iwt/ko/sup/EgovAriaInfo.do>.
- [42] ARIA source code - KISA. <http://seed.kisa.or.kr/iwt/ko/index.do;jsessionid=DDB1FEE78B04222BFCEA4E93A8264485>.

